Exp No:5

Solving Travelling Salesman Problem using Simulated Annealing

Srinidhi.V
18C102

Aim:

To solve travelling salesman problem using randomized search-technique like simulated annealing.

Algorithm:

* Simulated annealing is an optimization algorithm which is based on the metallurgical practices in which a material is heated to a high temperature and cooled.

* Define initial temperature and the minimum temperature. At each step, the current temperature is multiplied by some fraction alpha & decreased until it reaches the minimum temperature.

* For each distinct temperature value, find a neighbouring solution and accept with probability e^ (f(c) - f(n)) where c - current solution, n - neighbouring solution.

* Neighbouring solution is found by applying a slight permutation to the current solution.

* After mutating, calculate the change in e and if it is better than the current state, then update the state.

* When the minimum temperature is reached, return the best state.

* End.

18CL02

Sample Input And Output:

Cities ['New York', 'Los Angeles', 'Chicago', 'Minneapolis', 'Denver', 'Dallas', 'Seattle', 'Boston', 'San Francisco', 'St. Louis', 'Houston', 'Phoenix', 'Salt Lake City']

Start Location : Chicago.

Distance Matrix:

[[0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972], [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579], [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260], [1018, 1524, 355, 0, 700, 0, 663, 1021, 1769, 949, 796, 819, 586, 371], [1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 215, 887, 999], [2408, 959, 1737, 1395, 1021, 1081, 0, 2493, 678, 1724, 1891, 1194, 701] etc. ->.

Best solution using simulated annealing:

Chicago -> St. Louis -> Minneapolis -> Denver -> Salt Lake City -> Seattle -> San Francisco -> Los Angeles -> Phoenix -> Dallas -> Houston -> New York -> Boston -> Chicago.

Total Distance -> 7534 miles

Program:

```
# Import libraries
import sys
import random
import copy
import numpy as np
# This class represent a state
class State:
    # Create a new state
    def __init__(self, route:[], distance:int=0):
        self.route = route
        self.distance = distance
    # Compare states
    def __eq__(self, other):
        for i in range(len(self.route)):
            if(self.route[i] != other.route[i]):
                return False
        return True
    # Sort states
    def __lt__(self, other):
         return self.distance < other.distance
    # Print a state
    def __repr__(self):
        return ('({0},{1})\n'.format(self.route, self.distance))
    # Create a shallow copy
    def copy(self):
        return State(self.route, self.distance)
    # Create a deep copy
    def deepcopy(self):
        return State(copy.deepcopy(self.route), copy.deepcopy(self.distance))
    # Update distance
    def update_distance(self, matrix, home):

        # Reset distance
        self.distance = 0
        # Keep track of departing city
        from_index = home
        # Loop all cities in the current route
        for i in range(len(self.route)):
            self.distance += matrix[from_index][self.route[i]]
            from_index = self.route[i]
        # Add the distance back to home
```

```
        self.distance += matrix[from_index][home]
# This class represent a city (used when we need to delete cities)
class City:
    # Create a new city
    def __init__(self, index:int, distance:int):
        self.index = index
        self.distance = distance
    # Sort cities
    def __lt__(self, other):
        return self.distance < other.distance
# Return true with probability p
def probability(p):
    return p > random.uniform(0.0, 1.0)
# Schedule function for simulated annealing
def exp_schedule(k=20, lam=0.005, limit=1000):
    return lambda t: (k * np.exp(-lam * t) if t < limit else 0)
# Get the best random solution from a population
def get_random_solution(matrix:[], home:int, city_indexes:[], size:int,
use_weights:bool=False):
    # Create a list with city indexes
    cities = city_indexes.copy()
    # Remove the home city
    cities.pop(home)
    # Create a population
    population = []
    for i in range(size):
        if(use_weights == True):
            state = get_random_solution_with_weights(matrix, home)
        else:
            # Shuffle cities at random
            random.shuffle(cities)
            # Create a state
            state = State(cities[:])
            state.update_distance(matrix, home)
        # Add an individual to the population
        population.append(state)
    # Sort population
    population.sort()
    # Return the best solution
    return population[0]
# Get best solution by distance
def get_best_solution_by_distance(matrix:[], home:int):

    # Variables
```

```
    route = []
    from_index = home
    length = len(matrix) - 1
    # Loop until route is complete
    while len(route) < length:
         # Get a matrix row
        row = matrix[from_index]
        # Create a list with cities
        cities = {}
        for i in range(len(row)):
            cities[i] = City(i, row[i])
        # Remove cities that already is assigned to the route
        del cities[home]
        for i in route:
            del cities[i]
        # Sort cities
        sorted = list(cities.values())
        sorted.sort()
        # Add the city with the shortest distance
        from_index = sorted[0].index
        route.append(from_index)
    # Create a new state and update the distance
    state = State(route)
    state.update_distance(matrix, home)
    # Return a state
    return state
# Get a random solution by using weights
def get_random_solution_with_weights(matrix:[], home:int):

    # Variables
    route = []
    from_index = home
    length = len(matrix) - 1
    # Loop until route is complete
    while len(route) < length:
         # Get a matrix row
        row = matrix[from_index]
        # Create a list with cities
        cities = {}
        for i in range(len(row)):
            cities[i] = City(i, row[i])
        # Remove cities that already is assigned to the route
        del cities[home]
        for i in route:
```

```
            del cities[i]
        # Get the total weight
        total_weight = 0
        for key, city in cities.items():
            total_weight += city.distance
        # Add weights
        weights = []
        for key, city in cities.items():
            weights.append(total_weight / city.distance)
        # Add a city at random
        from_index = random.choices(list(cities.keys()), weights=weights)[0]
        route.append(from_index)
    # Create a new state and update the distance
    state = State(route)
    state.update_distance(matrix, home)
    # Return a state
    return state
# Mutate a solution
def mutate(matrix:[], home:int, state:State, mutation_rate:float=0.01):

    # Create a copy of the state
    mutated_state = state.deepcopy()
    # Loop all the states in a route
    for i in range(len(mutated_state.route)):
        # Check if we should do a mutation
        if(random.random() < mutation_rate):
            # Swap two cities
            j = int(random.random() * len(state.route))
            city_1 = mutated_state.route[i]
            city_2 = mutated_state.route[j]
            mutated_state.route[i] = city_2
            mutated_state.route[j] = city_1
    # Update the distance
    mutated_state.update_distance(matrix, home)
    # Return a mutated state
    return mutated_state
# Simulated annealing
def simulated_annealing(matrix:[], home:int, initial_state:State, mutation_rate:float=0.01,
schedule=exp_schedule()):

    best_state = initial_state

    for t in range(sys.maxsize):
```

```
        T = schedule(t)

        if T == 0:
            return best_state

        neighbor = mutate(matrix, home, best_state, mutation_rate)

        delta_e = best_state.distance - neighbor.distance

        if delta_e > 0 or probability(np.exp(delta_e / T)):
            best_state = neighbor
# The main entry point for this module
def main():
    # Cities to travel
    cities = ['New York', 'Los Angeles', 'Chicago', 'Minneapolis', 'Denver', 'Dallas', 'Seattle',
'Boston', 'San Francisco', 'St. Louis', 'Houston', 'Phoenix', 'Salt Lake City']
    city_indexes = [0,1,2,3,4,5,6,7,8,9,10,11,12]
    # Index of start location
    home = 2 # Chicago
    # Distances in miles between cities, same indexes (i, j) as in the cities array
    matrix = [[0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972],
            [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579],
            [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260],
            [1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280, 987],
            [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371],
            [1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999],
            [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701],
            [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099],
            [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600],
            [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162],
            [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200],
            [2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504],
            [1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0]]
    # Get the best route by distance
    state = get_best_solution_by_distance(matrix, home)
    print('-- Best solution by distance --')
    print(cities[home], end='')
    for i in range(0, len(state.route)):
      print(' -> ' + cities[state.route[i]], end='')
    print(' -> ' + cities[home], end='')
    print('\n\nTotal distance: {0} miles'.format(state.distance))
    print()
    # Get the best random route
    state = get_random_solution(matrix, home, city_indexes, 100)
```

```
    print('-- Best random solution --')
    print(cities[home], end='')
    for i in range(0, len(state.route)):
      print(' -> ' + cities[state.route[i]], end='')
    print(' -> ' + cities[home], end='')
    print('\n\nTotal distance: {0} miles'.format(state.distance))
    print()
    # Get a random solution with weights
    state = get_random_solution(matrix, home, city_indexes, 100, use_weights=True)
    print('-- Best random solution with weights --')
    print(cities[home], end='')
    for i in range(0, len(state.route)):
      print(' -> ' + cities[state.route[i]], end='')
    print(' -> ' + cities[home], end='')
    print('\n\nTotal distance: {0} miles'.format(state.distance))
    print()
    # Run simulated annealing to find a better solution
    state = get_best_solution_by_distance(matrix, home)
    state = simulated_annealing(matrix, home, state, 0.1)
    print('-- Simulated annealing solution --')
    print(cities[home], end='')
    for i in range(0, len(state.route)):
      print(' -> ' + cities[state.route[i]], end='')
    print(' -> ' + cities[home], end='')
    print('\n\nTotal distance: {0} miles'.format(state.distance))
    print()
# Tell python to run main method
if __name__ == "__main__": main()
```

Output:

```
In [1]: runfile('D:/18C102/Artificial Intelligence Lab/Simulated_Annealing_TSP_5.py', wdir='D:/18C102/
Artificial Intelligence Lab')
-- Best solution by distance --
Chicago -> St. Louis -> Minneapolis -> Denver -> Salt Lake City -> Phoenix -> Los Angeles -> San Francisco ->
Seattle -> Dallas -> Houston -> New York -> Boston -> Chicago

Total distance: 8131 miles

-- Best random solution --
Chicago -> Boston -> New York -> Phoenix -> Dallas -> Denver -> Houston -> St. Louis -> Minneapolis -> Los
Angeles -> San Francisco -> Seattle -> Salt Lake City -> Chicago

Total distance: 11349 miles

-- Best random solution with weights --
Chicago -> New York -> Boston -> Minneapolis -> Denver -> Phoenix -> Los Angeles -> Salt Lake City -> San
Francisco -> Seattle -> Dallas -> Houston -> St. Louis -> Chicago

Total distance: 8396 miles

-- Simulated annealing solution --
Chicago -> St. Louis -> Minneapolis -> Denver -> Salt Lake City -> Seattle -> San Francisco -> Los Angeles ->
Phoenix -> Dallas -> Houston -> New York -> Boston -> Chicago

Total distance: 7534 miles
```

Result:

Thus the travelling salesman problem is solved using randomized search technique simulated annealing.